Chapter 1 - Take the Red Pill

Transforming Data | Data should not change instead it should be transformed into new data. **Functions** are data transformers. Functions should be smaller and focused on doing one thing.

Parallel Processing | Elixir can run functions in parallel. It provides powerful mechanism for passing messages between them.

Installing Elixir (Ubuntu) | Installing Elixir

```
wget https://packages.erlang-solutions.com/erlang-solutions_1.0_all.deb
sudo dpkg -i erlang-solutions_1.0_all.deb
sudo apt-get update
sudo apt-get install esl-erlang
sudo apt-get install elixir
Terminal -> Elixir -v
Interactive Elixir -> iex
```

Chapter 2 - Pattern Matching

= a match operator | Assignment in Elixir is a pattern match. The left side of assignment (=) is called a **match clause** and right side of = is called a **value**. If both sides of = operator has same structure and each item in match clause can be matched with the corresponding term in the values then match is succeeded. This process is called **pattern matching**.

> ***Joe Armstrong, Erlang's creator, compares the equals sign in Erlang to that used in algebra. When you write the equation x = a + 1, you are not assigning the value of a + 1 to x. Instead you're simply asserting that the expressions x and a + 1 have the same value. If you know the value of x, you can work out the value of a, and vice versa.***

Chapter 3 - Immutability

All values are immutable | It simply means that all functions in Elixir, built in or the one you will write always return new copy/data of the data passed to it. Data or variables or values can not me changed outside the functions. We never capitalize a given string instead we will write a function that will take a string and return a new capitalized string.

Lots of processes | In Elixir you write your code using lots of processes. Each process has its own heap. These processes are very light weight and all data is discarded as function terminates. As a result garbage collector runs faster.

# Chapter 4 - Elixir Basics

Built-in Types:

Integer, Float, Atoms, Tuples, List, Keyword list, Map, PID, Ports, Binaries

Atoms | They are constants. Similar to Ruby symbols. There can be only one atom with same name across all modules.

- Atom name is its value i.e. :atom => atom
- Atom name always starts with :
- Atoms are not garbage collected so never generate them dynamically
- Atoms are labels, used to tag variables because they provide faster pattern matching
- Atoms are stored in Erlang Atom Table, Erlang VM supports 1,048,576 atoms

Ranges | As it's name indicates it is a range between start and end. Exp 1..100

PID | It is a id/reference of a local or remote process. PID of current process can be get by calling self. A new PID is created when we spawn a new process.

Tuples | It is a collection of values written between braces. It is common for functions to return tuple. In Elixir tuple is used for 3 to 4 values more than that map collection is recommended.

```
 {:user, "John", "registration"}
```

- Tuple represents a single piece of data that has a sementic meanings.
- Mostly used to return status code and data from a function, i.e. {:ok, data}
- Never iterate through tuple instead use Tuple.to_list
- Fetching tuple is fast but updating tupel is expensive
- Idiomatic way of getting Tuple elements are through pattern matching

```
{a, b, c} = {“elem1”, “elem2”, “elem3”}
```

Lists | It is a linked list. It is used when a collection has variable size and meant to grow or change.

- It is fast to iterate, fast to update but fetching a random element is slow
- It can have repeating elements

Keywordlist | It is a list of two element tuples where first element is an atom .

```
DB.save record, [ {:use_transaction, true}, {:logging, "HIGH"} ]
```

- It has become default mechanism for passing options to functions in Elixir
- It is an ordered list and allows duplicate keys
- When keyword list is last argument of a function then [] brackets are optional therefore they are

used to pass optional arguments as well

Maps | A map is a collection of key/value pairs

- It is not an ordered list and does not allow duplicate keys in a list
- If keys are atoms, you can also use . to access them
- A map is created using the %{} syntax:

```
map = %{:a => 1, 2 => :b}
```

# Chapter 5 - Anonymous Functions

Elixir is a functional language so functions are basics.

Anonymous functions do not have name but they are first class functions. They have function clause and function body. Function clause provide parameter list that is then matched through pattern inside the function body.

```
sum = fn (a,b) -> a + b end
sum.(1+2)

fun = fn ({:ok, file}) -> "Read data #{IO.read(file, :line)}"
```

Last value is returned by functions automatically. Functions can be passed as arguments and in Elixir this ability of function passing is used very commonly.

& = shortcut of fn | Short anonymous functions can be written with & notations. &1,&2.. are parameters.

```
add_one = &(&1+1)
```

Arity | Arity is the number of parameters of an existing function.

# Chapter 6 -Modules and Named Functions

Module |> In Elixir we write code into named functions. These named functions must be written in a module so a module is a collection of named functions.

```
#times.exs
defmodule Times do
  def double(n) do
    n * n
  end
  #single line function
  def speak(str), do: IO.puts str
end
#compile
iex "times.exs"
#call
Times.double(5)
```

Different functions with same names |> Multiple functions can have same name as far as they have different arity. Therefore a function is identified together with its name and with number of arity exp. double/1, double/3

- In an anonymous function if we have different match clauses (parameter list) we can write multiple match clauses for it. Same can be done with named functions but function will be written multiple times with different match clauses.
- Elixir will match functions with same names in the given order, if given arguments do not match with the parameter list then it tries next definition and do it until find the match or runs out of functions.
- Functions with same name must be defined adjacently in the source file.

```
 defmodule Factorial do
   def fac(0 ), do: 1
   def fac(n), do: n * fac(n-1)
 end
```

Recursive Functions |> These kind of functions are very common is Elixir. First look for the case that has a definitive answer i.e. **anchor** then find the recursive solution.

Guard Clauses |> As we know a function has a match clause that Elixir use to match with the given arguments. Now along with them we can have guard clause that can further filter the arguments depending on their values or given conditions. **when** keyword is used for this purpose.

Default Parameters |> Default parameters can be given to functions using \ value.

- If a function is called with less arguments then call will be failed
- If a function is called with exact number of arguments then arguments will be passed in same order

- If a function is called with exact number of required (non default) parameters then arguments will passed to them and default values will be used for others
- If a function is called with more than required values but less that total count of parameters then parameters are matched left to right.

```
def func(p1, p2 \\ 2, p3 \\ 3, p4) do
  IO.inspect [p1, p2, p3, p4]
end

Example.func("a", "b") # => ["a",2,3,"b"]
Example.func("a", "b", "c") # => ["a","b",3,"c"]
Example.func("a", "b", "c", "d") # => ["a","b","c","d"]
```

Private Functions |> **defp** macro define a private function. It can only be called within the module.

|> - The Amazing Pipe Operator |> The operator takes the results of the expression on the **left** and pass it as the **first** argument of the function to its right.

```
timesheet_approval = Timesheet.update_timesheet([timesheet_id: 123, startTime:
"2016-09-11 13:00",
endTime: "2016-09-11 17:00"])
|> calculate_weekly_hours
|> has_hours_exceeded
|> send_notifications
```

Module |> Module provides namesapces for the things you define. Nested modules can be used to improve redability and resuse.

- **import** directive is used to import module functions defined in it.
- **alias** directive creates an alias for a module.
- **require** directive make sure complete module is loaded before your code tires to use any macros defined in it.
- **Module attributes** starts with @. They can be access from inside the functions but only can be set outside the functions.

# Chapter 7 - Lists and Recursion

List can be empty [] or consists of head and tail. **Head** is the first element of the list and remaining list is called the **Tail**. Tail is also a list and both head and tail are separated with pipe **|** sign.

- List can be used in pattern matching

```
[head | tail ] = [1,2,3,4]
head = 1
tail = [2,3,4]
```

- Length of an empty list is 0
- In a recursive function if we have to remember a value of a variable between function calls then we can pass it as a function parameter because we can not hold them in a global variable as in Elixir values can not be changed or assigned outside the function .

# Chapter 8 - Dictionaries: Maps, HashDicts, Keywords, Sets, and Structs

Dictionary |> Dictionary is a **data type** that associate keys with values. **Maps**, **Keyword** list and **HashDict** are examples of dictionaries.

- We mostly use dictionary module to use its functionality so that we can change one type of dictionary to another for example map to hashdict.

```
#key word list to hashdict
[one: 1, two: 2, three: 3] |> Enum.into(HastDict.new) |> Dict.values |> Enum.sum
```

How to Choose Between Maps, HashDicts, and Keywords

- For duplicate keys use keyword list
- For ordered list use keyword
- Frequently accessing an element of list? Use maps
- Use HasDict instead of map if list have more than few hundred items**Updating a map** |> We can update key/value entries into the map using a pipe **|** sign

```
map = %{one: 1, two: 2, three: 3}
# add new keys
new_map = Dict.put(map,four,4)
# update values
new_map = %{map | two: 22}
```

Struct |> Struct are user defined maps. Sometimes we like to have a map with fixed keys and default values with pattern matching capabilities.

- It must be defined inside a module
- Keys must be atoms
- it cannot use Dict and Access module functionality
- **defstruct** macro is used to define struct
- They play a large role when implementing polymorphism

```
struct = %Subscriber{name: "Dave", over_18: true}
struct.name
```

# Chapter 9 - An Aside—What Are Types?

# Chapter 10 - Processing Collections - Enum and Stream

All **Collections** like tuple, list, keywordlist, map, HashDict, binaries, they share one thing that we can iterate through them. And for that purpose most of the time we will be using **Enum** and **Stream** modules.

Enum |> This is mostly used module in the Elixir world.

- Enum module can be used to iterate, filter, combine, split and to manipulate the collections

```
Enum.to_list 1..20
Enum.into [1,2,3], %{}
Enum.concat([1,2,3],[4,5,6])
Enum.map([1,2,3], &(&1 * &1))
Enum.filter([1,2,5,8], &(&1 > 4))
Enum.reduce([1,2,3], 0, &(&1 + &2))
```

Streams |> Stream module lets you iterate through a collection lazily. It means calculations will be only performed when a function is called. To get **results** from a stream, pass it as a parameter to the Enum function.

- Because we can pass a stream to a stream function therefore streams are composable
- We use streams when we need to deal a large numbers of things/data without necessarily generating them all at once.

```
s = Stream.map([1,2,3], &(&1 * 10))
Enum.to_list s

[1,2,3,4]
|> Stream.map(&(&1*&1))
|> Stream.map(&(&1+1))
|> Stream.filter(fn x -> rem(x,2) == 1 end)
|> Enum.to_list
```

Comprehensions |> It is an Elixir macro that lets you map and filter the collections at same time. Therefore it is a shortcout to these both functions.

```
for collection(s), filter(s), do: generate_new_collection
for x <- [1,2,3,4,5], x > 2 , do: x
```

- Any variable created in comprehension is available in rest of the compresion
- Compresion returns a list but it can be forced to return some other collection using Enum.into

```
 for x <- ~w{ cat dog }, into: %{"ant" => "ANT"}, do: { x, String.upcase(x) }
%{"ant" => "ANT", "cat" => "CAT", "dog" => "DOG"}
```

# Chapter 11 - Strings and Binaries

Elixir has two types of strings **String** and **Character list**.

String |> A double quoted string is called a string or more formally a binary.

Character List |> A single quoted string is actually a list of characters and behaves like a list. String module is not available for character lists. It can only work with strings.

Heredocs |> We can write a multi line strings using heredoc. It is mostly used for documentation.

Sigils |> Style literals in Elixir are called sigils. A sigil starts with ~ tilde character followed by a character.

```
# ~c = returns a character list
~c{a b c} => 'a b c'
# ~s = returns a string
~s{a b c} => "a b c"
~s{abc} => "abc"
# ~w = returns a list of whitespace-delemated words
~w{a b c} => ["a","b","c"]
```

Binaries |> It represents a sequence of bits. A binary literal looks like this << e,e,.. >>

Chapter 12 - Control Flow

- Combination of guard clauses, pattern matching and the fact that in elixir we write very short functions, replaces needs of control flow.
- Elixir provides if, unless, cond and case control flows.

if and unless |> if and unless are two functions that takes a condition and a keyword list as parameters. Keyword list includes do: and else: keys.

```
if 1 == 1, do: "true", else: "false"
unless 1 == 2, do: "true", else: "false"
```

cond |> Using cond we can list a series of conditions along with their codes. It executes the code of first truthy condition. It can be passed as a parameter.

- True is used to handle the situation when none of the above conditions are met.

```
time = 13
answer = cond do
           time == 11 -> "Good morning"
           time == 13 -> "Good afternoon"
           true -> "Time is #{time}"
         end
```

case |> In **cond** we provide a series of conditions but in **case** we provide a series of patterns. These patterns can have guard clauses.

```
open = File.open("README.md")
result = case open do
           {:ok, file} -> "File is opened successfully, #{IO.read(file,:line)}"
           {:error, reason} -> "Failed to open file reason = #{reason}"
         end
```

Raising Exceptions |> Raise an exception with a raise function.

```
raise "Raising an exception"
raise RuntimeError, "Error message"
```

- We use exceptions far less in Elixir than in other languages—the design philosophy is that errors should propagate back up to an external, supervising process.
- By convention in Elixir a function name with trailing exclamation (!) sign will raise a meaningful exception.

# Chapter 13 - Organizing a Project

Mix |> It is a Elixir project manager similar to ruby bundler. It is a command line utility.

```
# new project
mix new project_name
# install dependencies
# cd to project_name
mix deps.get
# run tests
mix test
# run project
iex -S mix
# recompile and run
mix run -e "MainModule.main_function"
```

lib/ By convention main module and all application code is stored here.

- create a sub directory with the main module name for example create a directory with name issues in lib/
- create issues.ex file in lib/
- All sub modules with in issues will be stored in issues directory
- Same is true for other nested modules

ex Vs exs |> Files with extension .ex are compiled files and files with .exs are directly converted into interpreted files. Interpreted files take longer to convert but they run faster.

- We create test files with .exs and source code files with .ex extensions

ExUnit|> An Elixir built in testing framework

```
defmodule IssuesTest do
  use ExUnit.Case

  test "test me" do
    assert 1==1
  end
end
```

Hex |> Hex is a elixir package manager. Elixir packages, applications, libraries are available on http://hex.pm

mix.exs |> It is elixir project's configuration file. It resides in the project folder. **mix.exs** has different functions that return a keyword list. Each keyword specified different settings.

```
  # mix.exs

  #project settings
```

```
    def project do
      [
        app: :issues,
        name: "Issues",
        version: "0.0.1",
        deps: deps
      ]

    #OTP framework settings
    def application do
      [
        application: [:logger, :htrtpoison]
      ]
    end

    #dependencies
    defp deps do
     {:poison, "~>0.4" }
    end
```

# Chapter 14 - Working with Multiple Processes

# Chapter 15 - Nodes - The Key to Distributing Services

Node |> As we know that Elixir runs on a Erlang VM. This VM is called a node.

- A computer can have more than one node
- Nodes can be given name to distinguish themselves over a network
- They can be connected with other nodes across the LAN or across the internet
- Node name is given when starting an iex session

```
# fully qualified complete name
iex --name node_one@machine_name
# short name
iex --sname node_one
iex> Node.self
# node name must be fully qualified and case sensitive
iex> Node.connect :"node_two@dev.local"
iex> Node.list
#run process on other node
iex> Node.spawn(:"node_two@dev.local",func)
```

Beam |> Elixir code is converted into a beam code. Erlang VM only understand beam code. Therefore Erlang VM is also referred as Beam.

Erlang VM |> It not only runs code plus it is a complete light weight OS running on top of the operating system. It has its own events, process scheduling, memory and inter process communication mechanism.

Node with cookie |> A node can only be connected to other node if cookies of both nodes match. Otherwise connection will be refused.

```
iex --sname one --cookie nodecookie
iex> Node.get_cookie
```

Named PID |> PID has three fields node number, low and high bits of the process ID. Exp. #PID<8966.59.0>

- PID can be given a global name using **:global.register_name** function
- We can list the name of all PIDs in mix.ex file that our application will register to avoid conflicting names already registered globally
- General rule is to register your process PIDs names when your application starts

```
func = fn -> IO.puts "anonymous function" end
pid = spawn(func)
:global.register_name(:pidname, pid)
pid = :global_whereis_name(:pidname)
send pid, {:ok, "Got you"}
```